Listing of Claims

The following listing of claims shall replace all prior listings and versions of claims in this application.

Listing of claims:

1. (Cancelled)

2. (Cancelled)

3. (Cancelled)

4. (Previously Presented) A method of verifying a program fragment downloaded onto a reprogrammable embedded system, equipped with a rewritable memory, a microprocessor and a virtual machine equipped with an execution stack and with operand registers, said program fragment consisting of an object code and including at least one subprogram consisting of a series of instructions manipulating said operand registers, said microprocessor and virtual machine configured to interpret said object code, said embedded system being interconnected to a reader, wherein subsequent to a detection of a downloading command and a storage of said object code in said rewritable memory, said method, for each subprogram, comprises:

initializing a type stack and a table of register types through data representing a state of said virtual machine on initialization of an execution of said temporarily stored object code;

carrying out a verification process of said temporarily stored object code instruction by instruction, by discerning an existence, for each current instruction, of a target, a branching-instruction target, a target of an exception-handler call or a target of a subroutine call, and, said current instruction being a target of a branching instruction, said verification process including verifying that said stack is empty and rejecting said program fragment otherwise;

verifying and updating an effect of said current instruction on said data types of said type stack and of said table of register types;

said verification process being successful when said table of register types is not modified in the course of a verification of all said instructions, and said verification process being carried out instruction by instruction until said table of register types is stable, with no modification being present, said verification process being interrupted and said program fragment being rejected, otherwise.

5. (Previously Presented) The method of claim 4, wherein said variable types which are manipulated during said verification process include at least:

class identifiers corresponding to object classes which are defined in said program fragment;

numeric variable types including at least a type for an integer coded on a given number of bits, designated as short type, and a type for the return address of a jump instruction, designated as a return address type;

references of null objects designated as null type;

object type relating to objects designated as object type;

a first specific type representing an intersection of all said types and corresponding to a zero value, designated as an intersection type;

a second specific type representing a union of all said types and corresponding to any type of value, designated as said union type.

6. (Previously Presented) The method of claim 5, wherein all said variable types verify a subtyping relation:

object type belongs to said union type;

short type and return address type belong to said union type; and

said intersection type belongs to null type, short type or return address type.

7. (Cancelled)

8. (Previously Presented) The method of claim 4, wherein said current instruction being a target of a subroutine call, said verification process comprising:

verifying that a previous instruction to said current instruction is an unconditional branching, a subroutine return or a withdrawal of an exception; and reupdating said stack of variable types by an entity of the return address type, formed by said return address of the subroutine, in case of a positive verification process; and,

rejecting said program fragment in case said verification process is failing, otherwise.

9. (Previously Presented) The method of claim 4, wherein said current instruction being a target of an exception handler, said verification process comprising:

verifying that a previous instruction to said current instruction is an unconditional branching, a subroutine return or a withdrawal of an exception; and

reupdating said type stack, by entering an exception type, in case of a positive verification process; and

rejecting said program fragment in case of said verification process is failing.

10. (Previously Presented) The method of claim 4, wherein said verification process fails and said program fragment is rejected if said current instruction is said target of multiple incompatible branchings.

11. (Previously Presented) The method of claim 4, wherein said verification process comprises continuing by passing to an update of said type stack if said current instruction is not said target of any branching.

12. (Previously Presented) The method of claim 4, wherein said verifying and updating includes, at least:

verifying that said type execution stack includes at least as many entries as a current instruction includes operands;

unstacking and verifying that types of entries at a top of said stack are subtypes of types of operands of said operands of said current instruction;

verifying an existence of a sufficient memory space on said type stack to proceed to stack said results of said current instruction;

stacking data types which are assigned to the results on said stack.

13. (Previously Presented) The method of claim 12, wherein said current instruction being an instruction to read a register of a given address, said verification process comprises:

verifying said data type of the result of a corresponding reading, by reading an entry at said given address in said table of register types;

determining said effect of said current instruction on said type stack by unstacking said entries of the stack corresponding to the operands of said current instruction and by stacking said data type of said result.

14. (Previously Presented) The method of claim 12, wherein said current instruction being an instruction to write to a register of a given address, said verification process comprises:

determining an effect of said current instruction on said type stack and said given type of said operand which is written in this register at said given address;

replacing said type entry of said table of register types at said given address by said type immediately above said previously stored type and above said given type of said operand which is written in said register at said given address.

15. (Previously Presented) A method of transforming an object code of a program fragment including a series of instructions, in which operands of each instruction belong to data types manipulated by said instruction, an execution stack does not exhibit any overflow phenomenon, and for each branching instruction, a type of stack variables at a corresponding branching is the same as that of targets of branching, into a standardized object code for this same program fragment, wherein, for all said instructions of said object code, said method comprising:

annotating each current instruction with a data type of said stack before and after execution of said current instruction, with said annotation data being calculated by means of an analysis of a data stream relating to said current instruction;

detecting, within said instructions and within each current instruction, an existence of branchings, or of branching-targets, for which said execution stack is not empty, said detecting operation being carried out on a basis of the annotation data of said type of stack variables allocated to each current instruction; and in case of detection of a non-empty execution stack,

inserting instructions to transfer stack variables on either side of said branchings or of said branching targets respectively, in order to empty contents of said execution stack into temporary registers before said

branching and to reestablish said execution stack from said temporary registers after said branching; and not inserting any transfer instruction otherwise, said method allowing to obtain a standardized object code for said same program fragment, in which said operands of each instruction belong to said data types manipulated by said instruction, said execution stack does not exhibit any overflow phenomenon, said execution stack is empty at each branching instruction and at each branching—target instruction, in an absence of any modification to the execution of said program fragment.

16. (Previously Presented) A method of transforming an object code of a program fragment including a series of instructions, in which operands of each instruction belong to data types manipulated by said instruction, and at least one of the operands of a given type are written into a register by an instruction of object code ~~is~~ reread from said register by another instruction of said object code with a same given data type, into a standardized object code for the program fragment, wherein for all said instructions of said object code, said method comprising:

annotating each current instruction with said data type of said registers before and after execution of said current instruction, with said annotation data being calculated by means of an analysis of a data stream relating to said instruction;

carrying out a reallocation of said registers, by detecting at least one of original registers employed with different types, dividing these original registers into separate standardized registers, with one standardized register for each data type used, and

reupdating said instructions which manipulate said operands which use said standardized registers;

said method configured to obtain said standardized object code for said program fragment in which said operands of each instruction belong to said data types manipulated by said instruction, said data type being allocated to a same register throughout said standardized object code.

17. (Previously Presented) The method of claim 15, wherein said detecting within said instructions and within each current instruction of an existence of branchings, or of branching targets, for which said execution stack is not empty, after detection of each corresponding instruction of given rank comprises:

associating with each of said instructions of said given rank a set of new registers, one of said new registers being associated with each said stack variable which is active at said instruction; and

examining each of said detected instructions of said given rank and discerning ~~the~~ an existence of a branching target or branching, and, in the case where said instruction of said given rank is a branching target and that said execution stack of said instruction is not empty,

for every preceding instruction, a preceding rank of said given rank and consisting of a branching, a withdrawal of an exception or a program return, said detected instruction of said given rank being accessible only by a branching, inserting a set of loading instructions to load from the set of new registers before said detected instruction of said given rank, with a

redirection of all branchings to said detected instruction of said given rank to a first inserted loading instruction; and

for every preceding instruction, of rank preceding said given rank, continuing in sequence, said detected instruction of said given rank being accessible simultaneously from a branching and from said preceding instruction of rank preceding said given rank, inserting a set of backup instructions to back up to a set of new registers before said detected instruction of said given rank, and a set of loading instructions to load from said set of new registers, with a redirection of all said branchings to said detected instruction of said given rank to said first inserted loading instruction, and, in the case where said detected instruction of said given rank is a branching to a given instruction,

for every detected instruction of said given rank consisting of an unconditional branching, inserting, before said detected instruction of said given rank, multiple backup instructions, a backup instruction being associated with each new register; and

for every detected instruction of said given rank consisting of a conditional branching instruction, and for a given number greater than zero of operands manipulated by said conditional branching instruction, inserting, before said detected instruction of said given rank, a permutation instruction, atop of said execution stack of said operands of said detected instruction of said given rank and at least one of a following values, a corresponding permutation operation adapted to collect at said top of said execution stack said following values to be backed up in said set of new registers; inserting, before said instruction of said given rank, a set of backup instructions to back up to said set of new registers; and inserting, after said detected instruction of said given rank, a set of load instructions to load from said set of new registers.

18. (Previously Presented) The method of claim 16, wherein carrying out reallocating registers by detecting said original register employed with different types comprises:

determining a lifetime interval of each register;

determining a main data type of each lifetime interval, said main data type of said lifetime interval for said given register being defined by the upper bound of said data types stored in said given register by said backup instructions belonging to said lifetime interval;

establishing an interference graph between said lifetime intervals, said interference graph consisting of a non-oriented graph of which each peak consists of a lifetime interval, and of which arcs between two peaks exist if one of said peaks contains a backup instruction addressed to said register of said other peak or vice versa;

translating a uniqueness of a data type which is allocated to each register in said interference graph, by adding arcs between all pairs of peaks of said interference graph while two peaks of a pair of peaks do not have a same associated main data type;

carrying out an instantiation of the interference graph, by assigning to each lifetime interval a register number, in such a way that different register numbers are assigned to two adjacent life time intervals in said interference graph.

19. (Cancelled)

20. (Previously Presented) An embedded system which can be reprogrammed by downloading program fragments, said embedded system including a least one microprocessor, one random-access memory, one input/output module, one electrically

reprogrammable nonvolatile memory and one permanent memory, an installed ~~a~~ main program and a virtual machine adapted to execute said installed main program and at least one program fragment using said microprocessor, wherein said embedded system includes at least one verification program module to verify a downloaded program fragment in accordance with a process comprising:

initializing a type stack and a table of register types through data representing a state of said virtual machine at a starting of an execution of said temporarily stored object code;

carrying out a verification process of said temporarily stored object code instruction by instruction, by discerning an existence, for each current instruction, of a target, a branching-instruction target, a target of an exception-handler call or a target of a subroutine call, and, said current instruction being said target of said branching instruction, said verification process consisting of verifying that said stack is empty and rejecting said program fragment otherwise;

carrying out a verification process and updating of an effect of said current instruction on said data types of said type stack and of said table of register types;

said verification process being successful when said table of register types is not modified in a course of a verification of all of said instructions, and said verification process being carried out instruction by instruction until said table of register types is stable, with no modification being present, said verification process being interrupted and said program fragment being rejected, otherwise;

said management and verification program module being installed in said permanent memory.

21. (Cancelled)

22. (Previously Presented) A system for transforming an object code of a program fragment including a series of instructions, in which operands of each instruction belong to data types manipulated by said instruction, an execution stack does not exhibit any overflow phenomenon and for each branching instruction, a type of stack variables at a corresponding branching is identical to the targets of this branching, and said operand, of given type, written to a register by an instruction of said object code is reread from said same register by another instruction of said object code with the same given data type, into a standardized object code for said same program fragment, wherein said transforming system includes, at least, installed in a working memory of a development computer or workstation, a program module for transforming said object code into a standardized object code in accordance with a process of transforming including for all said instructions of said object code comprising:

annotating each current instruction with said data type of said stack before and after execution of said current instruction, with an annotation data being calculated by means of analysis of the data stream relating to said current instruction;

detecting, within said instructions and within each current instruction, an existence of branchings, or respectively of branching-targets, for which said execution stack is not empty, said detecting operation being carried out on a basis of said annotation data of said type of stack variables allocated to each current instruction; and, in case of detection of a non—empty execution stack,

inserting instructions to transfer stack variables on either side of said branchings or of said branching targets respectively, in order to empty contents of said execution stack into temporary registers before said branching and to reestablish said execution stack from said temporary registers after said branching; and

not inserting any transfer instruction otherwise, said method allowing thus to obtain said standardized object code for said same program fragment, in which said operands of each instruction belong to said data types manipulated by said instruction, said execution stack does not exhibit any overflow phenomenon, said execution stack is empty at each branching instruction and at each branching-target instruction, in an absence of any modification to an execution of said program fragment.

23. (Cancelled)

24. (Previously Presented) A computer program product which is recorded on a medium and can be loaded directly from a terminal into an internal memory of a reprogrammable embedded system equipped with a microprocessor and a rewritable memory, said embedded system making it possible to download and temporarily store a program fragment consisting of an object code including a series of instructions, executable by said microprocessor by way of a virtual machine equipped with an execution stack and with operand registers manipulated via said instructions and making it possible to interpret said object code, said computer program product including portions of object code to execute at least one of steps of verifying a program fragment downloaded onto said embedded system according to a verifying process, said verifying process comprising:

initializing a type stack and a table of register types through data representing a state of said virtual machine at initialization of execution of said temporarily stored object code;

carrying out a verification process of said temporarily stored object code instruction by instruction, by discerning an existence, for each current instruction, of a target, a branching-instruction target, a target of an exception-handler call or a target of a subroutine call, and, said current instruction being a target of a branching instruction, said verification process consisting in verifying that said execution stack is empty and rejecting said program fragment otherwise;

carrying out a verification process and an updating of an effect of said current instruction on said data types of said type stack and of said table of register types;

said verification process being successful when said table of register types is not modified in a course of a verification of all said instructions, and said verification process being carried out instruction by instruction until said table of register types is stable, with no modification being present, said verification process being interrupted and said program fragment being rejected, otherwise.

25. (Previously Presented) A computer program product which is recorded on a medium including portions of object code to execute steps of a process of transforming an object code of a downloaded program fragment into a standardized object code for this same program, said process of transforming comprising:

annotating each instruction with a data type of a stack before and after execution of said current instruction, with said annotation data being calculated by means of an analysis of a data stream relating to said current instruction;

detecting, within said instructions and within each current instruction, an existence of branchings, or respectively of branching—targets, for which an execution stack is not empty, said detecting operation being carried out on based on said annotation data of a type of stack variables allocated to each current instruction, and, in case of detection of a non-empty execution stack;

inserting instructions to transfer stack variables on either side of said branchings or of said branching targets respectively, in order to empty contents of said execution stack into temporary registers before said branching and to reestablish the execution stack from said temporary registers after said branching; and

not inserting any transfer instruction otherwise, said method allowing thus to obtain said standardized object code for said same program fragment, in which the operands of each instruction belong to said data types manipulated by said instruction, said execution stack does not exhibit any overflow phenomenon, said execution stack is empty at each branching instruction and at each branching—target instruction, in absence of any modification to an execution of said program fragment.

26. (Previously Presented) A computer program product which is recorded on a medium and can be used in a reprogrammable embedded system, equipped with a microprocessor and a rewritable memory, said reprogrammable embedded system allowing to

download a program fragment consisting of an object code, a series of instructions, executable by said microprocessor of said reprogrammable embedded system by means of a virtual machine equipped with an execution stack and with local variables or registers manipulated via instructions and making it possible to interpret said object code, said computer program product comprising:

program resources which can be read by said microprocessor of said embedded system via said virtual machine, to command execution of a procedure for managing a downloading of a downloaded program fragment;

program resources which can be read by said microprocessor of said embedded system via said virtual machine, to command execution of a procedure for verifying, by instruction, said object code which makes up said program fragment;

program resources which can be read by said microprocessor of said embedded system via said virtual machine, to command execution of a downloaded program fragment subsequent to or in an absence of a conversion of said object code of said program fragment into a standardized object code for this same program fragment.

27. (Previously Presented) The computer program product as claimed in claim 26, additionally including at least one program resources which can be read by said microprocessor of said embedded system via said virtual machine, to command inhibition of execution, by said embedded system, of said program fragment in the case of an unsuccessful verification procedure of said program fragment.